

AD-A225 123

ANNUAL REPORT

VOLUME 1

TASK 1: DIGITAL EMULATION TECHNOLOGY LABORATORY

REPORT NO. AR-0142-90-001

July 22, 1990

DTIC
SELECTE
AUG 03 1990
S D

GUIDANCE, NAVIGATION AND CONTROL
DIGITAL EMULATION TECHNOLOGY LABORATORY

Contract No. DASG60-89-C-0142

Sponsored By

The United States Army Strategic Defense Command

COMPUTER ENGINEERING RESEARCH LABORATORY

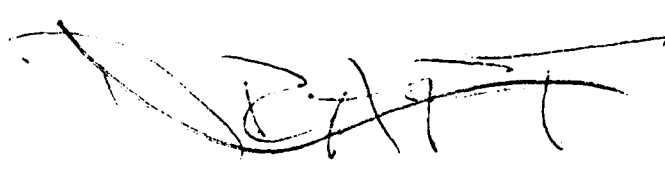
Georgia Institute of Technology

Atlanta, Georgia 30332 - 0540

Contract Data Requirements List Item A005

Period Covered: FY 90

Type Report: Annual


DISTRIBUTION STATEMENT A
Approved for public release
Distribution Unlimited

ANNUAL REPORT
VOLUME 1
TASK 1 DIGITAL EMULATION TECHNOLOGY LABORATORY

July 22, 1990

Author

Thomas R. Collins, Stephen R. Wachtel

COMPUTER ENGINEERING RESEARCH LABORATORY

Georgia Institute of Technology

Atlanta, Georgia 30332 - 0540

Eugene L. Sanders

USASDC

Contract Monitor

Cecil O. Alford

Georgia Tech

Project Director

Copyright 1990

Georgia Tech Research Corporation

Centennial Research Building

Atlanta, Georgia 30332

DISCLAIMER

DISCLAIMER STATEMENT - The views, opinions, and/or findings contained in this report are those of the author(s) and should not be construed as an official Department of the Army position, policy, or decision, unless so designated by other official documentation.

DISTRIBUTION CONTROL

- (1) **DISTRIBUTION STATEMENT** - Approved for public release; distribution is unlimited.
- (2) This material may be reproduced by or for the U.S. Government pursuant to the copyright license under the clause at DFARS 252.227 - 7013, October 1988.

Approved For	
DTIC	ORAS
DTIC	FAH
Unlimited	
Justification	
By	
Distribution	
Approved by	
Dist	Approved for
A-1	



1. Introduction

This report addresses the objectives, requirements, and schedule of the Digital Emulation Technology Laboratory (DETL), relative to contract number DASG60-89-C-0142. An associated report, "DETL Simulation Software (Old Contract)", covers DETL relative to contract number DASG60-85-C-0041. The major distinction between these two contracts and their associated activity at DETL is that this newer contract concerns primarily activity associated with the effort to develop an integrated hardware and software environment for end-to-end simulations of exoatmospheric interceptors such as EXOSIM. This includes the Georgia Tech Parallel Function Processor (PFP), system software for the PFP (utilities and parallel programming tools), and application software (EXOSIM). Some discussion of interfaces to specialized external hardware like the Seeker Signal Emulator (SSE) will also be included.

1.1. Objectives

Within DETL, there are two main hardware systems: the Parallel Function Processor (PFP) and the Seeker Scene Emulator (SSE). Each of these systems is a complex parallel processor, and they are designed to function together as an emulation facility for kinetic energy weapons systems. Software development is also an active area of research, both at the system level (compilers, loaders, graphics development) and at the application level (simulation and emulation studies).

The principal objectives of DETL are as follows:

- Provide facilities for 6-DOF KEW emulation,
- Provide real-time capability in excess of 2000 Hz,
- Provide real-time emulation of IR FPA seekers,
- Test and verify GN&C software and hardware systems, and
- Educate new PFP users and provide technical support.

The major components used in meeting these objectives include the PFP, SSE, high-speed 3-D graphics workstation, and associated conventional computers for basic support functions. Not all of these components are required for every task. For example, much of our work consists of running simulations (sometimes real-time, sometimes not) on the PFP, with no attached systems. This limited mode of operation is capable of verifying missile simulation models and control laws, as well as many types of signal processing.

To provide realistic imagery in real-time, however, the Seeker Scene Emulator is required. This system generates image data as though it were coming directly off of the elements of a focal-plane array, with the scene information determined by the relative location of the simulated missile system to the targets and decoys. Actual flight hardware may be tested within this

system. By equipping the hardware with appropriate interfaces to the PFP, the simulated functions of the GN&C Processor can migrate from the PFP to the actual hardware.

1.1.1. GN&C Test and Evaluation -- EXOSIM

The principle objective of DETL has always been to provide a facility in which guidance, navigation, and control algorithms can be run at high speeds in order to assess their performance. Recently, this has been served by implementing EXOSIM in various forms. EXOSIM is a simulation of a representative exoatmospheric interceptor (ERIS baseline) which has evolved from several earlier simulations, including KWEST and KEERIS. Unlike KWEST, which was written in a combination of ACSL and FORTRAN, EXOSIM is written entirely in FORTRAN. Unfortunately, the programming model for EXOSIM was not especially suited for a parallel implementation, since it utilized an event-driven structure. This technique is often used to enhance the performance of discrete-event simulations on single-processor systems, since it eliminates the need to model small increments of time in which essentially nothing changes. For a continuous system, however, there is little advantage in using an event-driven structure.

One of the subcontractors for this work (Dynetics) modified Version 1.0 of EXOSIM, changing it from an event-driven structure to a time-driven structure. At the same time, it was made into an unclassified version by replacing the data set and changing two routines. It is this modified version of EXOSIM that has been implemented at DETL. We generated a set of guidelines for partitioning FORTRAN code on the PFP and described a means of testing the partitions on a single-processor system. Following these guidelines, Dynetics first produced a first-stage boost version of the modified EXOSIM, partitioned for four processors. This program is called BOOST1. They then produced a first/second-stage boost version (BOOST2), partitioned for five processors. Both of these programs ran correctly on the PFP, requiring only a simple procedure of splitting up the main program along documented partitions and adding the appropriate communication instructions (which is an automated process).

BOOST2 has subsequently been altered at DETL in order to extract more parallelism, thus using more processors. At this time, a version runs on ? processors at a speed of ? times real time (slower than real time by that factor).

1.1.2. Education and Technical Support

Three major activities have taken place with regard to PFP education and technical support. First, a class was held in December 1989 in which seven students were taught the basics of PFP technology. The students were employees of USASDC and its contractors. The class included material on parallel processing fundamentals, the PFP model of parallelism, PFP hardware, the host operating system, and typical applications. Approximately half of the time was used for hands-on experience with the PFP.

The second support activity was the production of the PFP Technical Data Package. Our principle contribution to this effort was the software documentation (both system software and programming tools), as well as hardware documentation of recent modifications that were made to support the expanded memory and increased functionality of the current PFP host.

Finally, we have recently begun to organize a technical committee, the Parallel Simulation Technology Working Group. This group will include members from SDC-affiliated companies who will meet to discuss simulation techniques, general parallel programming topics, PFP issues, and ongoing SDC simulation work. The first meeting is tentatively scheduled for August 1990.

1.2. Schedules and milestones

As of July 1990, there are three 32-processor PFP systems available. One of these is currently undergoing a transition from an earlier configuration to our latest 3-processor-rack configuration, with 386-based processors to replace the original 286-based processors. The other two systems are the 286-based machine allocated for KDEC and the FPP-based machine for internal development of FPP/Sun host software. Not included in these 3 PFPs are a limited test PFP system and the prototype Multibus II PFP system. Since last year, we have taken our older 8086-based PFP out of service.

The FPP-based PFP and the KDEC PFP both include the basic packaging and power supplies to support expansion to 64-processor capability. The 386-based PFP may eventually be paired with the Multibus II PFP to produce a 64-processor hybrid system.

The major milestones completed over the period of this report are as follows:

- Integration of 386/12 processors into PFP, making the 286/12 processors available for the KDEC PFP,
- Transition to a new host operating system (RMX II), allowing greater memory accessibility, virtual terminal support, and other features,
- Development of utility software on the new system, replacing (and enhancing) basic functions for loading and starting programs,
- Development of programming tools for the new system, including a "make" utility for application maintenance,
- Development of parallel-processing support utilities, including one that analyzes variable usage across partitions and one that automatically generates communication calls,
- Development of libraries of communication procedures for processor-processor and processor-host interaction, providing uniform interfaces across several languages (C, Fortran, Pascal, and PL/M),
- Design and development of a new "piggyback" board to provide crossbar communication capability to the 386/12 boards through their iSBX interfaces,
- Design modifications to the Multibus Repeater boards to support expanded memory accessibility (16 MBytes per rack, over 48 MBytes per 32-processor system),

- Design modifications to the 286/12 processors to make them completely interchangeable with 386/12 boards in the new, expanded-memory configuration,
- System-level repackaging (racks, power supplies, cabling),
- Presentation of onsite education in PFP programming,
- Hardware and software documentation for the PFP Technical Data Package,
- Definition of basic rules for developing parallel applications in FORTRAN and in ACSL on conventional single-processor systems,
- Application of these rules in versions of EXOSIM by Dynetics, followed by successful porting of 4- and 5-processor versions to the PFP.

The most significant causes for delays during the past year have been

- The forced conversion to a more sophisticated operating system with associated hardware support for greater memory addressability,
- Incompatibility between new floating-point coprocessors and the original 80387 devices,
- Fortran compiler bugs, and
- Loss of support from subcontractor (Dynetics).

The first item above arose mainly from the complexity of EXOSIM, but it has had beneficial side effects. The coprocessor and compiler problems were basically just unforeseen technical problems. These first three items caused only temporary delays can now be considered to be closed issues with no long-term impact to our schedule. The last item (loss of support from Dynetics) came about from a lack of funding. This is the major reason that we have not progressed beyond a second-stage implementation of EXOSIM, and it has impacted our schedule.

During the coming year, the highest priority will be placed on the optimal use of these technologies in simulation applications such as EXOSIM. The schedule for this effort is shown in Figure 1.1, and the individual milestones are described in Figure 1.2. The implementation dates for midcourse and end-to-end EXOSIM have changed from earlier estimates because of the reasons stated above. Dates for LEAP and GBI implementation are estimates and assume that these simulations are available to be ported to the PFP.

Figure 1.2 Task 1 Milestones

1. EXOSIM Version 1.0 boost phase installed and running on PFP with Seeker Emulator (Simple Target Model)
2. EXOSIM version 1.0 midcourse installed as supplied by other Government Contractor
3. EXOSIM Version 1.0 end-to-end available
4. EXOSIM Version 1.0 end-to-end installed and running on upgraded PFP
5. Draper Laboratory IMU Electronics Test Plan Complete
6. IMU Electronics Testing
7. IMU Test Report Complete
8. AHAT Processor Test Plan Complete
9. AHAT Processor Testing
10. AHAT Test Report Complete
11. LEAP 6-DOF Available
12. LEAP 6-DOF Installed
13. LEAP 6-DOF Running
14. GBI 6-DOF Available
15. GBI 6-DOF Installed
16. GBI 6-DOF Running

2. PFP Development Tools

2.1. Introduction

The PFP development tools consist of a collection of programs developed at the Georgia Institute of Technology. These tools, which execute on a Intel 310AP computer running iRMX II, assist the user in controlling the PFP hardware. Refer to Appendix ? for the source code listings for each program.

2.2. Reset

The reset program is used to reset the PFP hardware. Reset writes to an i/o port which causes the PFP to perform a hardware reset on the crossbar/sequencer and all processing elements. The command line syntax is:

```
reset
```

2.3. Download

The download program is used to download programs into the appropriate elements on the PFP. Download uses the first and second fields of each line in the input file to determine which elements to download and with which programs to download into the elements. The command line syntax is:

```
download <process.txt>
```

where:

<process.txt> = input file name.

2.4. Start

The start program is used to start the appropriate elements on the PFP. Start uses the first field of each line in the input file to determine which elements to start. The command line syntax is:

```
start <process.txt>
```

where:

<process.txt> = input file name.

2.5. Ioserve

The ioserve program is designed to handle any input or output between the host processor and any of the target processors. Ioserve uses the input file to determine whether or not a processing element will need any input by examining the third field in each line of the input file and whether or not a processing element will produce any output by examining the fourth field in each line of the input file. The command line syntax is:

`ioserve <process.txt> <timeout>`

where:

`<process.txt>` = input file name.

`<timeout>` = integer timeout count.

If the third field contains a character string other than '<NULL>' the string is assumed to be the name of the input file associated with that processing element. If the fourth field contains a character string other than '<NULL>' the string is assumed to be the name of the output file associated with that processing element. Since neither the crossbar nor sequencer support input and output, the third and fourth fields in the input file for these elements always contains '<NULL>'.

If the third field of the input file indicates a processing element requires input, ioserve will open the input file, read the data, and send it to the processing element at the beginning of the execution. If the fourth field of the input file contains a file name, ioserve writes the output from the processing element to that file. The output is always written to the terminal whether or not an output file is designated.

Each processing element can have unique input and output files or a combination of shared input and output files. If an input file is shared, each of the processing elements sharing the file should expect the same data as input. If an output file is shared, the output from all of those processing elements will be intermixed in the output file as it is processed by ioserve. Since this is a parallel environment, care will need to be taken when generating output as the order of the output can not be guaranteed.

Ioserve in turn scans the data available port for each of the active processing elements by opening the shared memory window to the processor and checking the appropriate flag. When data is available, ioserve retrieves the data and writes it to the designated output. If no data is available, ioserve closes the window and proceeds onto the next processing element. Ioserve retrieves data from a processing element until the source is exhausted. The second parameter on the command line is an integer timeout count in seconds. Ioserve scans the active processing elements for output until there has been no output for the specified number of seconds and then ioserve terminates.

2.6. Make

Make was originally developed as a project control tool for the UNIX operating system. In UNIX, as in iRMX, most programs are composed of many small source modules that need to be combined together to produce an executable module. Without a utility such as make, it would be necessary for the programmer to keep track of all object modules which might need to be regenerated due to changes in source files. The make program provides an easy way to automate this process. The command line syntax is:

`make <makefile>`

where:

<makefile> = input file name.

Make reads commands from a user-defined "makefile" that lists the files to be created, the commands that create them, and the files from which they are created. When you direct make to create a program, it makes sure that each file on which the program depends is up to date, then creates the program by executing the given commands. If a file is not up to date, make updates it before creating the program by executing explicitly given commands or one of the many built-in commands.

3. PFP Diagnostic Tools

3.1. Introduction

The PFP diagnostic tools consist of a collection of programs developed at the Georgia Institute of Technology. These tools, which execute on a Intel 310AP computer running iRMX II, assist the user in diagnosing the PFP. Refer to Appendix ? for the source code listings for each program.

3.2. Mtest

The mtest program is used to test processor, crossbar or sequencer memory to determine whether its working correctly. The command line syntax is:

```
mtest <name> <offset> <length> <test>
```

where:

<name> = processor, crossbar or sequencer name.

<offset> = hexadecimal memory offset.

<length> = hexadecimal memory length.

<test> = simple PATTERN test or complex RANDOM test.

The pattern test purpose is to determine whether each memory byte can correctly turn on and off. This test uses four steps:

- 1) write 00H into each byte and then read back to verify correctness.
- 2) write 55H into each byte and then read back to verify correctness.
- 3) write AAH into each byte and then read back to verify correctness.
- 4) write FFH into each byte and then read back to verify correctness.

The random test purpose is to determine whether each memory location can correctly be addressed. This test uses two passes:

- 1) generate a random number for each byte of memory and then write it to memory.
- 2) regenerate the random number for each byte of memory and then read the memory to verify correctness.

Even though the random test may seem simpler than the pattern test, the time required to generate the random numbers causes the random test to take 4 times longer than the pattern test on the iRMX II system.

3.3. Ntest

The ntest program is used to test processor, crossbar and sequencer communication to determine whether its working correctly. This test is uses a special program for each processor and a network program automatically generated from the user supplied process.txt file. A makefile is included to automate the command sequence.

3.4. Display

The display program is used to display processor, crossbar or sequencer memory in a hexadecimal and ascii formatted dump. The command line syntax is:

```
display <name> <offset> <length>
```

where:

<name> = processor, crossbar or sequencer name.

<offset> = hexadecimal memory offset.

<length> = hexadecimal memory length.

3.5. Fill

The fill program is used to fill processor, crossbar or sequencer memory with the specified byte. The command line syntax is:

```
fill <name> <offset> <length> <byte>
```

where:

<name> = processor, crossbar or sequencer name.

<offset> = hexadecimal memory offset.

<length> = hexadecimal memory length.

<byte> = hexadecimal byte.

3.6. Replace

The replace program is used to replace processor, crossbar or sequencer memory with the specified word(s). The command line syntax is:

```
replace <name> <offset> <length> <word> ... <word>
```

where:

<name> = processor, crossbar or sequencer name.

<offset> = hexadecimal memory offset.

<length> = hexadecimal memory length.

<word> = hexadecimal word.

4. Crossbar/Sequencer Code Generation Tools

4.1. Introduction

The crossbar/sequencer code generation tools consist of a collection of programs developed at the Georgia Institute of Technology. These tools, which execute on a DEC MVAX II running Ultrix 1, assist the user in constructing correct crossbar/sequencer code. Figure ? shows the relationship between each program and how they interact.

4.2. Type

The type program parses a complete FORTRAN main program, which includes both declarative and executable statements, and extracts data variable explicit and implicit type information. The command line syntax is:

```
type <input >output
```

where:

```
input = input file name.
```

```
output = output file name.
```

4.3. Usage

The usage program parses a partitioned block of executable statements from the FORTRAN main program and extracts data variable usage information. In addition, FORTRAN source for each subroutine called must be included in the input so that usage may determine how each subroutine parameter is used. The command line syntax is:

```
usage <input >output
```

where:

```
input = input file name.
```

```
output = output file name.
```

Usage parses the entire input file first and builds data structures which represent subprogram blocks and the variable usage within them. Then usage propagates subroutine formal argument variable usage to the actual argument variable usage through calls. This process continues recursively until no more variable usage propagation is required.

4.4. Combine

The combine program combines the type program output with up to 32 usage program outputs to form a output file which contains only the typed data variables which are used on more than one processor partition. The command line syntax is:

```
combine <input '<name0>=<block0>' ... '<name31>=<block31>' >output
```

where:

input = input file name.

'<name_i>=<block_i>' = processor name & partition name

output = output file name.

The combine program first builds a symbol table containing variable name, data type and array dimensions. Then combine reads each usage program output file and builds a data structure which represents the variables usage for each processor. Finally, combine outputs only those variables, with the associated data type and dimensions, which are used on more than one processor partition.

4.5. Sequence

The sequence program converts the output of the combine program output into crossbar/sequencer code and generates include files for each processor partition that contain corresponding send and receives in FORTRAN. The command line syntax is:

```
sequence <input '<name0>=<block0>' ... '<name31>=<block31>' >output
```

where:

input = input file name.

'<name_i>=<block_i>' = processor name & partition name

output = output file name.

The sequence programs reads the combine program output and builds a data structure which represents variable usage for each processor. The data structure is then sorted so as the variables that are used on the most processors are at the top of the list. Starting at the top of the list, the crossbar/sequencer code is generated for a variable and output. Also, the associated include file for each one of the processors using this variable is updated with the appropriate send or receive. The rest of the list is then searched from top to bottom to determine whether one or more variables may be sent during the same crossbar/sequencer cycle. If any are found then they are output and marked as output. Then the whole process repeats until all variables have been output.

4.6. Summary

The summary program converts the output of the combine program output into a summary table which shows the typed data variables usage which are used on more than one processor. The command line syntax is:

```
summary <input '<name0>=<block0>' ... '<name31>=<block31>'>output
```

where:

input = input file name.

'<name_i>=<block_i>' = processor name & partition name

output = output file name.

The combine programs reads the combine program output and then outputs a table in sorted order with variables listed down the page and processors listed across the page. On the last page, the number of input and outputs per processor is given. This number represents the theoretical lower limit on the number of crossbar/sequencer cycles required to implement these communications.

5. PFP Programming Library

In order to isolate the PFP programmer from the intricacies of interacting directly with the hardware, several libraries of routines have been developed. The most extensive of these libraries deal with target-target communication and with target-host communication, but there are also libraries that support high-accuracy timing and the control of LED indicators on each target board. In each case, separate interfaces have been developed for each of four languages: C, FORTRAN, Pascal, and PL/M. The first three languages are considered essential because of their wide usage, and PL/M is included because of its availability and efficient implementation on the Intel host system. Much effort has been made to make each language interface similar, but the detailed syntax of each language is necessarily different because of the languages themselves.

The two main types of communication (target-target and target-host) both allow for a wide range of message types (integers, strings, real numbers, arrays, etc.). The first section that follows will describe these message types in detail, as they relate to each of the four languages. This information thus applies to two later sections, one on target-target communication and one on target-host communication. These two sections describe the actual communication routine interfaces for each of the four languages in turn. Finally, this material concludes with information on the LED and timer libraries.

5.1. Message Types

The next four subsections describe the message types for C, FORTRAN, Pascal, and PL/M. The main content for each subsection is a table that lists the supported message types (in the third column of the table) along with the corresponding "native" standard data type for that particular language (in the first column). For example, the message type "LOGICAL_08BIT" (an eight-bit logical variable) corresponds to a "char" in C, a "logical*1" in FORTRAN, a "boolean" in Pascal, and a "byte" in PL/M. These are the most reasonable mappings that are possible between the various languages. For most of the languages, it is possible (and perhaps more convenient) to use an alternate data type in preference to the standard data type. For example, a communication routine in a C program that is designed to send CHARACTER_08BIT messages will function equally well with a variable of the standard C data type "char" or the alternate "CHARACTER_08BIT_type." This is strictly a matter of user preference. FORTRAN does not support the declaration of user-defined data types, so only the standard data types are available in FORTRAN programs.

Complex numbers are defined as structures (or records) in all languages but FORTRAN, which supports complex variables directly. All other types are simple data types, not structures. In all cases, arrays are handled as repetitions of the given data types. This will be made clearer in the examples that follow later.

The "network" column in each table gives an example of the crossbar code corresponding to a target-target communication of the given data type, assuming that a single variable of that type is sent from processor p00 to p01. The only difference in these lines of crossbar code is the

replication factor which appears after the decimal point. Since the crossbar allows for a maximum of 16-bit wide transfers during each cycle, it is necessary to use two cycles to send a 32-bit variable such as a single-precision real number. For this reason, each "network" entry for a single-precision real transfer will show "p01 := p00.2" to indicate that two cycles are required to send such a variable from p00 to p01. Similarly, some variables will require four cycles.

The application of each of these message types will be illustrated by examples in later sections.

5.1.1. C

The C message types are given in Table 1. Note that the type "char" can correspond to either the CHARACTER_08BIT message type or the LOGICAL_08BIT message, since C does not implement logical types directly. The standard data type "int" is avoided in preference to "short" and "long," which are of known length on any present or future host or target processor system.

Table 1: C message types

data_type	alternate data_type	message_type	network
char	CHARACTER_08BIT_type	CHARACTER_08BIT	p01 := p00.1
struct { float r; float i; }	COMPLEX_32BIT_type	COMPLEX_32BIT	p01 := p00.4
struct { double r; double i; }	COMPLEX_64BIT_type	COMPLEX_64BIT	p01 := p00.8
char	LOGICAL_08BIT_type	LOGICAL_08BIT	p01 := p00.1
short	LOGICAL_16BIT_type	LOGICAL_16BIT	p01 := p00.1
long	LOGICAL_32BIT_type	LOGICAL_32BIT	p01 := p00.2
float	REAL_32BIT_type	REAL_32BIT	p01 := p00.2
double	REAL_64BIT_type	REAL_64BIT	p01 := p00.4
signed char	SIGNED_08BIT_type	SIGNED_08BIT	p01 := p00.1
signed short	SIGNED_16BIT_type	SIGNED_16BIT	p01 := p00.1
signed long	SIGNED_32BIT_type	SIGNED_32BIT	p01 := p00.2
unsigned char	UNSIGNED_08BIT_type	UNSIGNED_08BIT	p01 := p00.1
unsigned short	UNSIGNED_16BIT_type	UNSIGNED_16BIT	p01 := p00.1
unsigned long	UNSIGNED_32BIT_type	UNSIGNED_32BIT	p01 := p00.2

5.1.2. FORTRAN

The FORTRAN message types are given in Table 2. As mentioned earlier, note that no alternate data types are available because of limitations within FORTRAN.

Table 2: FORTRAN message types.

data_type	alternate data_type	message_type	network
character		CHARACTER_08BIT	p01 := p00.1
complex*8		COMPLEX_32BIT	p01 := p00.4
complex*16		COMPLEX_64BIT	p01 := p00.8
logical*1		LOGICAL_08BIT	p01 := p00.1
logical*2		LOGICAL_16BIT	p01 := p00.1
logical*4		LOGICAL_32BIT	p01 := p00.2
real*4		REAL_32BIT	p01 := p00.2
real*8		REAL_64BIT	p01 := p00.4
integer*1		SIGNED_08BIT	p01 := p00.1
integer*2		SIGNED_16BIT	p01 := p00.1
integer*4		SIGNED_32BIT	p01 := p00.2
integer*1		UNSIGNED_08BIT	p01 := p00.1
integer*2		UNSIGNED_16BIT	p01 := p00.1
integer*4		UNSIGNED_32BIT	p01 := p00.2

5.1.3. Pascal

The Pascal message types are given in Table 3. Both signed and unsigned 8-bit variables map into the standard enumerated type [0..255], which is essentially a "byte" type. The communication routines will correctly interpret the byte variable for the user.

Table 3: Pascal message types

data_type	alternate data_type	message_type	network
char	CHARACTER_08BIT_type	CHARACTER_08BIT	p01 := p00.1
record r: real; i: real; end	COMPLEX_32BIT_type	COMPLEX_32BIT	p01 := p00.4
record r: longreal; i: longreal; end	COMPLEX_64BIT_type	COMPLEX_64BIT	p01 := p00.8
boolean	LOGICAL_08BIT_type	LOGICAL_08BIT	p01 := p00.1
integer	LOGICAL_16BIT_type	LOGICAL_16BIT	p01 := p00.1
longint	LOGICAL_32BIT_type	LOGICAL_32BIT	p01 := p00.2
real	REAL_32BIT_type	REAL_32BIT	p01 := p00.2
longreal	REAL_64BIT_type	REAL_64BIT	p01 := p00.4
0..255	SIGNED_08BIT_type	SIGNED_08BIT	p01 := p00.1
integer	SIGNED_16BIT_type	SIGNED_16BIT	p01 := p00.1
longint	SIGNED_32BIT_type	SIGNED_32BIT	p01 := p00.2
0..255	UNSIGNED_08BIT_type	UNSIGNED_08BIT	p01 := p00.1
word	UNSIGNED_16BIT_type	UNSIGNED_16BIT	p01 := p00.1
longint	UNSIGNED_32BIT_type	UNSIGNED_32BIT	p01 := p00.2

5.1.4. PL/M

The PL/M message types are given in Table 4. Both signed and unsigned 8-bit variables map into the standard "byte" type, as do 8-bit logical variables. Similarly, the standard "dword" type is used for 32-bit signed, unsigned, and logical variables, and the standard "word" type is used for 16-bit unsigned and logical variables. The communication routines will correctly interpret these variables for the user. PL/M does not support double-precision (64-bit) real variables.

Table 4: PL/M message types.

data_type	alternate data_type	message_type	network
byte	CHARACTER_08BIT_type	CHARACTER_08BIT	p01 := p00.1
structure (r real, i real)	COMPLEX_32BIT_type	COMPLEX_32BIT	p01 := p00.4
		COMPLEX_64BIT	
byte	LOGICAL_08BIT_type	LOGICAL_08BIT	p01 := p00.1
word	LOGICAL_16BIT_type	LOGICAL_16BIT	p01 := p00.1
dword	LOGICAL_32BIT_type	LOGICAL_32BIT	p01 := p00.2
real	REAL_32BIT_type	REAL_32BIT	p01 := p00.2
		REAL_64BIT	
byte	SIGNED_08BIT_type	SIGNED_08BIT	p01 := p00.1
integer	SIGNED_16BIT_type	SIGNED_16BIT	p01 := p00.1
dword	SIGNED_32BIT_type	SIGNED_32BIT	p01 := p00.2
byte	UNSIGNED_08BIT_type	UNSIGNED_08BIT	p01 := p00.1
word	UNSIGNED_16BIT_type	UNSIGNED_16BIT	p01 := p00.1
dword	UNSIGNED_32BIT_type	UNSIGNED_32BIT	p01 := p00.2

5.2. Target Processor to Target Processor Communication

Having a basic understanding of the variable types, we can now turn to the communication routines, beginning with the target-target routines. For each message type, there are complementary send and receive routines. Any of these can be used on any target processor, as long as the complementary routine is included at the appropriate place in the program running on the "connected" target (as defined by the crossbar and sequencer code). These routines cannot be used on the host processor -- they are for sending and receiving through the crossbar network only. These routines operate only on single scalar (or structure) variables. Arrays are handled by repeating the communication routine as many times as necessary. (This differs from the built-in repetition capability of the host-target communication routines which will be discussed later.)

5.2.1. Send

Each of the send routines are described by example in the four sections which follow (one section for each language). The receive routines will be described afterwards.

5.2.1.1. C

Refer to Appendix ? for a complete listing of the source code for these routines. Example 1 illustrates each of the send routines. Note that these routines expect a pointer to their only parameter, accomplished in these examples by using the "&" operator.

Example 1: C Send

```

COMPLEX_32BIT_type c32;
COMPLEX_64BIT_type c64;
LOGICAL_08BIT_type l08;          /* or "char 108;" */
LOGICAL_16BIT_type l16;          /* or "short 108;" */
LOGICAL_32BIT_type l32;          /* or "long 108;" */
REAL_32BIT_type r32;             /* or "float r32;" */
REAL_64BIT_type r64;             /* or "double r32;" */
SIGNED_08BIT_type s08;           /* and so on ... */
SIGNED_16BIT_type s16;
SIGNED_32BIT_type s32;
UNSIGNED_08BIT_type u08;         /* and so on ... */
UNSIGNED_16BIT_type u16;
UNSIGNED_32BIT_type u32;

send_COMPLEX_32BIT( &c32 );
send_COMPLEX_64BIT( &c64 );

send_LOGICAL_08BIT( &l08 );
send_LOGICAL_16BIT( &l16 );
send_LOGICAL_32BIT( &l32 );

send_REAL_32BIT( &r32 );
send_REAL_64BIT( &r64 );

send_SIGNED_08BIT( &s08 );
send_SIGNED_16BIT( &s16 );
send_SIGNED_32BIT( &s32 );

send_UNSIGNED_08BIT( &u08 );
send_UNSIGNED_16BIT( &u16 );
send_UNSIGNED_32BIT( &u32 );

```

5.2.1.2. FORTRAN

Refer to Appendix ? for a complete listing of the source code for these routines. Example 2 illustrates each of the send routines. Note that, like any other FORTRAN subroutines, these routines require a "call" statement. FORTRAN defaults to passing parameters with pointers, which is the correct mode for these subroutines.

Example 2: FORTRAN Send.

```

COMPLEX*8      c32
COMPLEX*16     c64
LOGICAL*1      108
LOGICAL*2      116
LOGICAL*4      132
REAL*4  r32
REAL*8  r64
INTEGER*1      s08
INTEGER*2      s16
INTEGER*4      s32
INTEGER*1      u08
INTEGER*2      u16
INTEGER*4      u32

call send_COMPLEX_32BIT( c32 )
call send_COMPLEX_64BIT( c64 )

call send_LOGICAL_08BIT( 108 )
call send_LOGICAL_16BIT( 116 )
call send_LOGICAL_32BIT( 132 )

call send_REAL_32BIT( r32 )
call send_REAL_64BIT( r64 )

call send_SIGNED_08BIT( s08 )
call send_SIGNED_16BIT( s16 )
call send_SIGNED_32BIT( s32 )

call send_UNSIGNED_08BIT( u08 )
call send_UNSIGNED_16BIT( u16 )
call send_UNSIGNED_32BIT( u32 )

```

5.2.1.3. Pascal

Refer to Appendix ? for a complete listing of the source code for these routines. Example 3 illustrates each of the send routines. Pascal also defaults to the correct mode of passing parameters by location (with pointers).

Example 3: Pascal Send.

```

c32 : COMPLEX_32BIT_type;
c64 : COMPLEX_64BIT_type;
108 : LOGICAL_08BIT_type; { or "108 : boolean;" }
116 : LOGICAL_16BIT_type; { or "116 : integer;" }
132 : LOGICAL_32BIT_type; { or "132 : longint;" }
r32 : REAL_32BIT_type; { and so on ... }
r64 : REAL_64BIT_type;
s08 : SIGNED_08BIT_type;
s16 : SIGNED_16BIT_type;
s32 : SIGNED_32BIT_type;
u08 : UNSIGNED_08BIT_type;
u16 : UNSIGNED_16BIT_type;
u32 : UNSIGNED_32BIT_type;

send_COMPLEX_32BIT( c32 );
send_COMPLEX_64BIT( c64 );

send_LOGICAL_08BIT( 108 );
send_LOGICAL_16BIT( 116 );
send_LOGICAL_32BIT( 132 );

send_REAL_32BIT( r32 );
send_REAL_64BIT( r64 );

send_SIGNED_08BIT( s08 );
send_SIGNED_16BIT( s16 );
send_SIGNED_32BIT( s32 );

send_UNSIGNED_08BIT( u08 );
send_UNSIGNED_16BIT( u16 );
send_UNSIGNED_32BIT( u32 );

```

5.2.1.4. PL/M

Refer to Appendix ? for a complete listing of the source code for these routines. Example 4 illustrates each of the send routines. Note that these routines expect a pointer to their only parameter, which must be explicitly coded with the "@" operator.

Example 4: PL/M Send.

```
declare c32 COMPLEX_32BIT_type;
declare l08 LOGICAL_08BIT_type;
declare l16 LOGICAL_08BIT_type;
declare l32 LOGICAL_08BIT_type;
declare r32 REAL_32BIT_type;
declare s08 SIGNED_08BIT_type;
declare s16 SIGNED_16BIT_type;
declare s32 SIGNED_32BIT_type;
declare u08 UNSIGNED_08BIT_type;
declare u16 UNSIGNED_16BIT_type;
declare u32 UNSIGNED_32BIT_type;

call send_COMPLEX_32BIT( @c32 );

call send_LOGICAL_08BIT( @l08 );
call send_LOGICAL_16BIT( @l16 );
call send_LOGICAL_32BIT( @l32 );

call send_REAL_32BIT( @r32 );

call send_SIGNED_08BIT( @s08 );
call send_SIGNED_16BIT( @s16 );
call send_SIGNED_32BIT( @s32 );

call send_UNSIGNED_08BIT( @u08 );
call send_UNSIGNED_16BIT( @u16 );
call send_UNSIGNED_32BIT( @u32 );
```

5.2.2. Receive

Each of the receive routines are described by example in the four sections which follow (one section for each language). The general form of these routines is identical to the complementary send routines.

5.2.2.1. C

Refer to Appendix ? for a complete listing of the source code for these routines. Example 5 illustrates each of the receive routines. Note that these routines use the "&" operator to pass a pointer.

Example 5: C Receive.

```

COMPLEX_32BIT_type c32;
COMPLEX_64BIT_type c64;
LOGICAL_08BIT_type l08;          /* or "char 108;" */
LOGICAL_16BIT_type l16;          /* or "short 108;" */
LOGICAL_32BIT_type l32;          /* or "long 108;" */
REAL_32BIT_type r32;             /* or "float r32;" */
REAL_64BIT_type r64;             /* or "double r32;" */
SIGNED_08BIT_type s08;           /* and so on ... */
SIGNED_16BIT_type s16;
SIGNED_32BIT_type s32;
UNSIGNED_08BIT_type u08;
UNSIGNED_16BIT_type u16;
UNSIGNED_32BIT_type u32;

receive_COMPLEX_32BIT( &c32 );
receive_COMPLEX_64BIT( &c64 );

receive_LOGICAL_08BIT( &l08 );
receive_LOGICAL_16BIT( &l16 );
receive_LOGICAL_32BIT( &l32 );

receive_REAL_32BIT( &r32 );
receive_REAL_64BIT( &r64 );

receive_SIGNED_08BIT( &s08 );
receive_SIGNED_16BIT( &s16 );
receive_SIGNED_32BIT( &s32 );

receive_UNSIGNED_08BIT( &u08 );
receive_UNSIGNED_16BIT( &u16 );
receive_UNSIGNED_32BIT( &u32 );

```

5.2.2.2. FORTRAN

Refer to Appendix ? for a complete listing of the source code for these routines. Example 6 illustrates each of the receive routines. Note that, like any other FORTRAN subroutines, these routines require a "call" statement. Again, FORTRAN defaults to passing parameters by location.

Example 6: FORTRAN Receive.

```

COMPLEX*8      c32
COMPLEX*16     c64
LOGICAL*1      108
LOGICAL*2      116
LOGICAL*4      132
REAL*4  r32
REAL*8  r64
INTEGER*1      s08
INTEGER*2      s16
INTEGER*4      s32
INTEGER*1      u08
INTEGER*2      u16
INTEGER*4      u32

call receive_COMPLEX_32BIT( c32 )
call receive_COMPLEX_64BIT( c64 )

call receive_LOGICAL_08BIT( 108 )
call receive_LOGICAL_16BIT( 116 )
call receive_LOGICAL_32BIT( 132 )

call receive_REAL_32BIT( r32 )
call receive_REAL_64BIT( r64 )

call receive_SIGNED_08BIT( s08 )
call receive_SIGNED_16BIT( s16 )
call receive_SIGNED_32BIT( s32 )

call receive_UNSIGNED_08BIT( u08 )
call receive_UNSIGNED_16BIT( u16 )
call receive_UNSIGNED_32BIT( u32 )

```

5.2.2.3. Pascal

Refer to Appendix ? for a complete listing of the source code for these routines. Example 7 illustrates each of the receive routines. Again, Pascal defaults to passing parameters by location.

Example 7: Pascal Receive.

```

c32 : COMPLEX_32BIT_type;
c64 : COMPLEX_64BIT_type;
108 : LOGICAL_08BIT_type; { or "108 : boolean;" }
116 : LOGICAL_16BIT_type; { or "116 : integer;" }
132 : LOGICAL_32BIT_type; { or "132 : longint;" }
r32 : REAL_32BIT_type;    { and so on ... }
r64 : REAL_64BIT_type;
s08 : SIGNED_08BIT_type;
s16 : SIGNED_16BIT_type;
s32 : SIGNED_32BIT_type;
u08 : UNSIGNED_08BIT_type;
u16 : UNSIGNED_16BIT_type;
u32 : UNSIGNED_32BIT_type;

receive_COMPLEX_32BIT( c32 );
receive_COMPLEX_64BIT( c64 );

receive_LOGICAL_08BIT( 108 );
receive_LOGICAL_16BIT( 116 );
receive_LOGICAL_32BIT( 132 );

receive_REAL_32BIT( r32 );
receive_REAL_64BIT( r64 );

receive_SIGNED_08BIT( s08 );
receive_SIGNED_16BIT( s16 );
receive_SIGNED_32BIT( s32 );

receive_UNSIGNED_08BIT( u08 );
receive_UNSIGNED_16BIT( u16 );
receive_UNSIGNED_32BIT( u32 );

```

5.2.2.4. PL/M

Refer to Appendix ? for a complete listing of the source code for these routines. Example 8 illustrates each of the receive routines. Note that these routines use the "@" operator to pass a pointer.

Example 8: PL/M Receive.

```

declare c32 COMPLEX_32BIT_type;
declare l08 LOGICAL_08BIT_type;
declare l16 LOGICAL_16BIT_type;
declare l32 LOGICAL_32BIT_type;
declare r32 REAL_32BIT_type;
declare s08 SIGNED_08BIT_type;
declare s16 SIGNED_16BIT_type;
declare s32 SIGNED_32BIT_type;
declare u08 UNSIGNED_08BIT_type;
declare u16 UNSIGNED_16BIT_type;
declare u32 UNSIGNED_32BIT_type;

call receive_COMPLEX_32BIT( @c32 );

call receive_LOGICAL_08BIT( @l08 );
call receive_LOGICAL_16BIT( @l16 );
call receive_LOGICAL_32BIT( @l32 );

call receive_REAL_32BIT( @r32 );

call receive_SIGNED_08BIT( @s08 );
call receive_SIGNED_16BIT( @s16 );
call receive_SIGNED_32BIT( @s32 );

call receive_UNSIGNED_08BIT( @u08 );
call receive_UNSIGNED_16BIT( @u16 );
call receive_UNSIGNED_32BIT( @u32 );

```

5.3. Target Processor to Host Computer Communication

It is often necessary to provide for host interaction with the target processors. Typically, this is to provide initial data values (from host to targets) or to output run-time values (from targets to the host disk or to the host console). Two libraries of routines are provided for this purpose. The "Input" routines are used for receiving messages at either a host or a target, and the "Output" routines are used to send messages from a host or a target. We avoid the use of the terms "send" and "receive" from now on, reserving them to describe crossbar communication (as in the previous examples). Although the actual routines that are linked into a host program differ from the same routines linked into a target program, the user interface is identical, presenting a simple, uniform interface to the programmer. Although target I/O operations must match up with complementary host I/O operations, it is possible to write programs in a way that is relatively tolerant of sequencing variations, as opposed to the corresponding operations over the crossbar network. One convenient way to do this is to not write a custom host program for each application, using instead the "IOserver" program described earlier in this manual. The IOserver, after initialization, continually polls all active targets to determine if any are attempting to send messages to the host. These messages are passed to the console or to a disk file.

5.3.1. Input

The input libraries are provided to transfer messages TO the calling program. As before, each of the libraries is broken down into a subsection of examples for each language. The `message_size` parameter allows the routine to repetitively input variables of the given type, which is useful for array input (including character strings). In each of the examples below, note that all three parameters (including `message_type` and `message_size`) are outputs from each routine. This means that the message type and size are transmitted to the inputting processor along with the

message itself and allows for applications that may or may not know what type of messages are forthcoming. If the message type is unknown, the message should be stored in a generic array of bytes, with type conversion to be performed later. In the examples, however, we assume that a known type of message is being input and thus assign the incoming message to a variable of the appropriate type. We further assume, just as an example, that the message length is less than or equal to 4, declaring the variable arrays accordingly.

5.3.1.1. C

Refer to Appendix ? for a complete listing of the source code for these routines. Example 9 illustrates each of the input routines. Note that these routines expect a pointer to each of the three parameters.

Example 9: C Input.

```
SIGNED_16BIT_type message_type;
SIGNED_16BIT_type message_size;
COMPLEX_32BIT_type c32[4];
COMPLEX_64BIT_type c64[4];
LOGICAL_08BIT_type l08[4];
LOGICAL_16BIT_type l16[4];
LOGICAL_32BIT_type l32[4];
REAL_32BIT_type r32[4];
REAL_64BIT_type r64[4];
SIGNED_08BIT_type s08[4];
SIGNED_16BIT_type s16[4];
SIGNED_32BIT_type s32[4];
UNSIGNED_08BIT_type u08[4];
UNSIGNED_16BIT_type u16[4];
UNSIGNED_32BIT_type u32[4];

input_message( &message_type, &c32, &message_size );
input_message( &message_type, &c64, &message_size );

input_message( &message_type, &l08, &message_size );
input_message( &message_type, &l16, &message_size );
input_message( &message_type, &l32, &message_size );

input_message( &message_type, &r32, &message_size );
input_message( &message_type, &r64, &message_size );

input_message( &message_type, &s08, &message_size );
input_message( &message_type, &s16, &message_size );
input_message( &message_type, &s32, &message_size );

input_message( &message_type, &u08, &message_size );
input_message( &message_type, &u16, &message_size );
input_message( &message_type, &u32, &message_size );
```

5.3.1.2. FORTRAN

Refer to Appendix ? for a complete listing of the source code for these routines. Example 10 illustrates each of the input routines. All values are passed by location (with pointers).

Example 10: FORTRAN Input.

```

INTEGER*2 message_type
INTEGER*2 message_size
COMPLEX*8      c32(4)
COMPLEX*16     c64(4)
LOGICAL*1      108(4)
LOGICAL*2      116(4)
LOGICAL*4      132(4)
REAL*4  r32(4)
REAL*8  r64(4)
INTEGER*1  s08(4)
INTEGER*2  s16(4)
INTEGER*4  s32(4)
INTEGER*1  u08(4)
INTEGER*2  u16(4)
INTEGER*4  u32(4)

call input_message( message_type, c32, message_size )
call input_message( message_type, c64, message_size )

call input_message( message_type, 108, message_size )
call input_message( message_type, 116, message_size )
call input_message( message_type, 132, message_size )

call input_message( message_type, r32, message_size )
call input_message( message_type, r64, message_size )

call input_message( message_type, s08, message_size )
call input_message( message_type, s16, message_size )
call input_message( message_type, s32, message_size )

call input_message( message_type, u08, message_size )
call input_message( message_type, u16, message_size )
call input_message( message_type, u32, message_size )

```

5.3.1.3. Pascal

Refer to Appendix ? for a complete listing of the source code for these routines. Example 11 illustrates each of the input routines. Again, all values are passed by location (with pointers).

Example 11: Pascal Input.

```

message_type: SIGNED_16BIT_type;
message_size: SIGNED_16BIT_type;
c32 : array [1..4] of COMPLEX_32BIT_type;
c64 : array [1..4] of COMPLEX_64BIT_type;
l08 : array [1..4] of LOGICAL_08BIT_type;
l16 : array [1..4] of LOGICAL_16BIT_type;
l32 : array [1..4] of LOGICAL_32BIT_type;
r32 : array [1..4] of REAL_32BIT_type;
r64 : array [1..4] of REAL_64BIT_type;
s08 : array [1..4] of SIGNED_08BIT_type;
s16 : array [1..4] of SIGNED_16BIT_type;
s32 : array [1..4] of SIGNED_32BIT_type;
u08 : array [1..4] of UNSIGNED_08BIT_type;
u16 : array [1..4] of UNSIGNED_16BIT_type;
u32 : array [1..4] of UNSIGNED_32BIT_type;

input_message( message_type, c32, message_size );
input_message( message_type, c64, message_size );

input_message( message_type, l08, message_size );
input_message( message_type, l16, message_size );
input_message( message_type, l32, message_size );

input_message( message_type, r32, message_size );
input_message( message_type, r64, message_size );

input_message( message_type, s08, message_size );
input_message( message_type, s16, message_size );
input_message( message_type, s32, message_size );

input_message( message_type, u08, message_size );
input_message( message_type, u16, message_size );
input_message( message_type, u32, message_size );

```

5.3.1.4. PL/M

Refer to Appendix ? for a complete listing of the source code for these routines. Example 12 illustrates each of the input routines. The three parameters are passed as pointer through the use of the "@" operator.

Example 12: PL/M Input.

```

declare message_type SIGNED_16BIT_type;
declare message_size SIGNED_16BIT_type;

declare c32(4) COMPLEX_32BIT_type;
declare l08(4) LOGICAL_08BIT_type;
declare l16(4) LOGICAL_08BIT_type;
declare l32(4) LOGICAL_08BIT_type;
declare r32(4) REAL_32BIT_type;
declare s08(4) SIGNED_08BIT_type;
declare s16(4) SIGNED_16BIT_type;
declare s32(4) SIGNED_32BIT_type;
declare u08(4) UNSIGNED_08BIT_type;
declare u16(4) UNSIGNED_16BIT_type;
declare u32(4) UNSIGNED_32BIT_type;

call input_message( @message_type, @c32, @message_size );

call input_message( @message_type, @l08, @message_size );
call input_message( @message_type, @l16, @message_size );
call input_message( @message_type, @l32, @message_size );

call input_message( @message_type, @r32, @message_size );

call input_message( @message_type, @s08, @message_size );
call input_message( @message_type, @s16, @message_size );
call input_message( @message_type, @s32, @message_size );

call input_message( @message_type, @u08, @message_size );
call input_message( @message_type, @u16, @message_size );
call input_message( @message_type, @u32, @message_size );

```

5.3.2. Output

The output libraries are provided to transfer messages FROM the calling program. As before, each of the libraries is broken down into a subsection of examples for each language. The `message_size` parameter allows the routine to repetitively output variables of the given type, which is useful for array output (including character strings). In each of the examples below, note that all three parameters (including `message_type` and `message_size`) are inputs to each routine. This means that the calling program can declare the message variable arrays of the appropriate length for the maximum-size message to be sent. We further assume, just as an example, that the message length is equal to 4, declaring the variable arrays accordingly.

Note that these routines expect a pointer to only the variable used to store the message. The other parameters are passed by value.

5.3.2.1. C

Refer to Appendix ? for a complete listing of the source code for these routines. Example 13 illustrates each of the output routines.

Example 13: C Output.

```

COMPLEX_32BIT_type c32[4];
COMPLEX_64BIT_type c64[4];
LOGICAL_08BIT_type l08[4];
LOGICAL_16BIT_type l16[4];
LOGICAL_32BIT_type l32[4];
REAL_32BIT_type r32[4];
REAL_64BIT_type r64[4];
SIGNED_08BIT_type s08[4];
SIGNED_16BIT_type s16[4];
SIGNED_32BIT_type s32[4];
UNSIGNED_08BIT_type u08[4];
UNSIGNED_16BIT_type u16[4];
UNSIGNED_32BIT_type u32[4];

output_message( COMPLEX_32BIT, &c32, 4 );
output_message( COMPLEX_64BIT, &c64, 4 );

output_message( LOGICAL_08BIT, &l08, 4 );
output_message( LOGICAL_16BIT, &l16, 4 );
output_message( LOGICAL_32BIT, &l32, 4 );

output_message( REAL_32BIT, &r32, 4 );
output_message( REAL_64BIT, &r64, 4 );

output_message( SIGNED_08BIT, &s08, 4 );
output_message( SIGNED_16BIT, &s16, 4 );
output_message( SIGNED_32BIT, &s32, 4 );

output_message( UNSIGNED_08BIT, &u08, 4 );
output_message( UNSIGNED_16BIT, &u16, 4 );
output_message( UNSIGNED_32BIT, &u32, 4 );

```

5.3.2.2. FORTRAN

Refer to Appendix ? for a complete listing of the source code for these routines. Example 14 illustrates each of the output routines. Note that it is necessary to force FORTRAN to pass constants by value.

Example 14: FORTRAN Output.

```

INTEGER*2 message_type
INTEGER*2 message_size
COMPLEX*8      c32(4)
COMPLEX*16     c64(4)
LOGICAL*1      l08(4)
LOGICAL*2      l16(4)
LOGICAL*4      l32(4)
REAL*4  r32(4)
REAL*8  r64(4)
INTEGER*1  s08(4)
INTEGER*2  s16(4)
INTEGER*4  s32(4)
INTEGER*1  u08(4)
INTEGER*2  u16(4)
INTEGER*4  u32(4)

call output_message( %VAL(COMPLEX_32BIT), c32, %VAL(4) )
call output_message( %VAL(COMPLEX_64BIT), c64, %VAL(4) )

call output_message( %VAL(LOGICAL_08BIT), l08, %VAL(4) )
call output_message( %VAL(LOGICAL_16BIT), l16, %VAL(4) )
call output_message( %VAL(LOGICAL_32BIT), l32, %VAL(4) )

call output_message( %VAL(REAL_32BIT), r32, %VAL(4) )
call output_message( %VAL(REAL_64BIT), r64, %VAL(4) )

call output_message( %VAL(SIGNED_08BIT), s08, %VAL(4) )
call output_message( %VAL(SIGNED_16BIT), s16, %VAL(4) )
call output_message( %VAL(SIGNED_32BIT), s32, %VAL(4) )

call output_message( %VAL(UNSIGNED_08BIT), u08, %VAL(4) )
call output_message( %VAL(UNSIGNED_16BIT), u16, %VAL(4) )
call output_message( %VAL(UNSIGNED_32BIT), u32, %VAL(4) )

```

5.3.2.3. Pascal

Refer to Appendix ? for a complete listing of the source code for these routines. Example illustrates each of the output routines.

Example : Pascal Output.

```

c32 : array [1..4] of COMPLEX_32BIT_type;
c64 : array [1..4] of COMPLEX_64BIT_type;
l08 : array [1..4] of LOGICAL_08BIT_type;
l16 : array [1..4] of LOGICAL_16BIT_type;
l32 : array [1..4] of LOGICAL_32BIT_type;
r32 : array [1..4] of REAL_32BIT_type;
r64 : array [1..4] of REAL_64BIT_type;
s08 : array [1..4] of SIGNED_08BIT_type;
s16 : array [1..4] of SIGNED_16BIT_type;
s32 : array [1..4] of SIGNED_32BIT_type;
u08 : array [1..4] of UNSIGNED_08BIT_type;
u16 : array [1..4] of UNSIGNED_16BIT_type;
u32 : array [1..4] of UNSIGNED_32BIT_type;

output_message( COMPLEX_32BIT, c32, 4 );
output_message( COMPLEX_64BIT, c64, 4 );

output_message( LOGICAL_08BIT, l08, 4 );
output_message( LOGICAL_16BIT, l16, 4 );
output_message( LOGICAL_32BIT, l32, 4 );

output_message( REAL_32BIT, r32, 4 );
output_message( REAL_64BIT, r64, 4 );

output_message( SIGNED_08BIT, s08, 4 );
output_message( SIGNED_16BIT, s16, 4 );
output_message( SIGNED_32BIT, s32, 4 );

output_message( UNSIGNED_08BIT, u08, 4 );
output_message( UNSIGNED_16BIT, u16, 4 );
output_message( UNSIGNED_32BIT, u32, 4 );

```

5.3.2.4. PL/M

Refer to Appendix ? for a complete listing of the source code for these routines. Example 16 illustrates each of the output routines.

Example 16: PL/M Output.

```

declare c32(4) COMPLEX_32BIT_type;
declare l08(4) LOGICAL_08BIT_type;
declare l16(4) LOGICAL_16BIT_type;
declare l32(4) LOGICAL_32BIT_type;
declare r32(4) REAL_32BIT_type;
declare s08(4) SIGNED_08BIT_type;
declare s16(4) SIGNED_16BIT_type;
declare s32(4) SIGNED_32BIT_type;
declare u08(4) UNSIGNED_08BIT_type;
declare u16(4) UNSIGNED_16BIT_type;
declare u32(4) UNSIGNED_32BIT_type;

call output_message( COMPLEX_32BIT, @c32, 4 );

call output_message( LOGICAL_08BIT, @l08, 4 );
call output_message( LOGICAL_16BIT, @l16, 4 );
call output_message( LOGICAL_32BIT, @l32, 4 );

call output_message( REAL_32BIT, @r32, 4 );

call output_message( SIGNED_08BIT, @s08, 4 );
call output_message( SIGNED_16BIT, @s16, 4 );
call output_message( SIGNED_32BIT, @s32, 4 );

call output_message( UNSIGNED_08BIT, @u08, 4 );
call output_message( UNSIGNED_16BIT, @u16, 4 );
call output_message( UNSIGNED_32BIT, @u32, 4 );

```

5.4. Target Processor Hardware

Two libraries are provided for support of onboard target-processor features. These two features are the real-time timer and the LEDs along the edge of the board.

5.4.1. Real-Time Timer

The real-time timer is convenient for measuring the elapsed time required for the execution of various routines. It may also be used to deliberately slow down a simulation that is running faster than real-time, providing a synchronizing mechanism with the real world. As always, these routines are accessible from each of the four languages to be described in the following sections.

There are only two timer routines. One resets the timer to zero, and the other returns the timer value (in ticks of the clock). To convert the timer value to seconds, divide the value by the timer clock frequency, 1.229 MHz. It is necessary to reset the timer at least once, but it may then be used for repeated timer reads (lap timing).

5.4.1.1. C

Refer to Appendix ? for a complete listing of the source code for these routines. Example 17 illustrates each of the real-time timer routines.

Example 17: C real-time timer.

```
UNSIGNED_32BIT_type u32;  
reset_timer( );  
u32 = read_timer( );
```

5.4.1.2. FORTRAN

Refer to Appendix ? for a complete listing of the source code for these routines. Example 18 illustrates each of the real-time timer routines.

Example 18: FORTRAN real-time timer.

```
INTEGER*4 u32  
call reset_timer( )  
u32 = read_timer( )
```

5.4.1.3. Pascal

Refer to Appendix ? for a complete listing of the source code for these routines. Example 19 illustrates each of the real-time timer routines.

Example 19: Pascal real-time timer.

```
u32 : UNSIGNED_32BIT_type;
reset_timer;
u32 := read_timer;
```

5.4.1.4. PL/M

Refer to Appendix ? for a complete listing of the source code for these routines. Example 20 illustrates each of the real-time timer routines.

Example 20: PL/M real-time timer.

```
declare u32 UNSIGNED_32BIT_type;
call reset_timer;
u32 = read_timer( );
```

5.4.2. LED

Two of the LEDs on the edge of the board are accessible by user programs. Both of these LEDs are red, and taken together they can be viewed as a binary number over the range 0-3 (ON-LED = 1, OFF-LED = 0). One LED routine is provided, and the only parameter is a value over this range. Examples for each language follow.

5.4.2.1. C

Refer to Appendix ? for a complete listing of the source code for these routines. Example 21 illustrates each of the LED routines.

Example 21: C LED.

```
led( 0 );
led( 1 );
led( 2 );
led( 3 );
```


5.4.2.2. FORTRAN

Refer to Appendix ? for a complete listing of the source code for these routines. Example 22 illustrates each of the LED routines.

Example 22: FORTRAN LED.

```
call led( %VAL(0) )  
call led( %VAL(1) )  
call led( %VAL(2) )  
call led( %VAL(3) )
```

5.4.2.3. Pascal

Refer to Appendix ? for a complete listing of the source code for these routines. Example 23 illustrates each of the LED routines.

Example 23: Pascal LED.

```
led( 0 );  
led( 1 );  
led( 2 );  
led( 3 );
```

5.4.2.4. PL/M

Refer to Appendix ? for a complete listing of the source code for these routines. Example 24 illustrates each of the LED routines.

Example 24: PL/M LED.

```
call led( 0 );  
call led( 1 );  
call led( 2 );  
call led( 3 );
```

6. Education and Technical Support

This section includes an overview of the materials provided for the PFP class which was held in December 1989 for potential PFP users at other sites, as well as a technical report on the use of ACSL as a parallel programming language. For other related technical support information, see the "PFP Technical Data Package" and the "PFP Programmer's Reference Manual," both of which were issued during the past year for this contract.

6.1. PFP Class Materials

The following section has been condensed from the presentation material actually used in the classroom sessions. It begins with some fundamentals about parallel computers, the PFP, and the partitioning of typical problems. Not included here is the class material on diagnostics, use of the host operating system, and laboratory exercises. Much of this omitted material is included elsewhere, either in the "PFP Technical Data Package" or the "PFP Programmer's Reference Manual."

Types of Parallel Computers

SISD/SIMD/MIMD designations

Programming models:

Control-driven (imperative)

Demand-driven (applicative, or functional)

Dataflow

Pattern-Driven

Types of Interprocessor Communication

Fixed network/Switchable network

Shared memory/Private Memory

Synchronous/Asynchronous

Characteristics of the Parallel Function Processor (PFP)

Control-driven

Switchable network

Private memory

Synchronous communication

Control-driven parallel machines

Most "natural" parallelization of serial uniprocessor architectures

Requires explicit processes for sending and receiving

All processes (functions, subroutines, assignments) ordered on all processors

Incorrect ordering of communication results in deadlock

Simple Example

$a = (b+c)/(d+e)$

Processor 1:

```
{ previous instructions }
temp := b+c
send (temp, processor3)
{ more instructions }
```

Processor 2:

```
{ previous instructions }
temp := d+e
send (temp, processor3)
{ more instructions }
```

Processor 3:

```
{ previous instructions }
receive(sum1, processor1)
receive(sum2, processor2)
a := sum1/sum2
{ more instructions }
```

Important aspects of Simple Example

Variable names on different processors have no correlation (private memory)

Each processor's code looks like a typical imperative language, except for a few sends and receives

Communication processes include a source or destination in this basic model -- but not in the PFP

In some parallel control-driven machines, the ordering of the receives on Processor 3 would make no difference -- but it does in the PFP

Effect of the Crossbar/Sequencer

The switching network provides the correct connections in a specific ordering (when properly programmed)

Each processor "sees" only one input/output port, which is connected to the right processor at the right time

There is no need to specify source or destination in sends and receives

The orderings of all communications on all processors must be compatible with the "master" ordering of the crossbar/sequencer

Typically, multiple "conversations" will take place during a single crossbar/sequencer cycle

Whenever a processor encounters a receive instruction, it must wait until the particular communication takes place (synchronous architecture), and the communication will not take place until all conversations are ready to proceed

Local buffering smooths out some of this effect

Deterministic physical systems

No effect occurs without cause

It is possible to identify subsystems with limited interactions (causes and effects)

Subsystems generally have some concept of memory (state)

For dynamic mechanical systems, the relevant states are positions, velocities, and forces (Newton)

We cannot overemphasize the concept of state. Assigning a "small" finite number of state variables in a system model is an approximation (there are a myriad of subatomic particles ultimately responsible for the "true" behavior), but it is generally possible to make reasonable models (relatively few states) based on physical principles.

Sometimes the states are easier to think of as continuous, other times discrete. External interactions are usually approximated, the result being state variables that aren't technically state variables (they have new values available at each time interval, but not as the result of integrating anything). Often it is possible to choose either a "pseudo-state" model or a true state model for an external input, such as a sine-wave generator. In this particular case we could just produce the pseudo-state $x = A \sin(\omega t) + \phi$ OR we could implement a second-order differential equation (with no external inputs, just two initial conditions to make the phase and amplitude right).

State Variables

All memory of the system is contained in some set of state variables

For a given system, there is no unique assignment of state variables, but there is a fixed number of state variables

State variables (and only state variables) are initialized

Continuous (analog) state variables (and only such state variables) are integrated each integration timestep

Discrete (sampled) state variables (and only such state variables) are updated each corresponding sample time

Input functions ($u=f(t)$) can sometimes be treated much like state variables, but without initialization or integration (i.e., we tend to describe "external" subsystems as functions of time, rather than explicitly modelling them)

Example: A Digital Logic Simulation

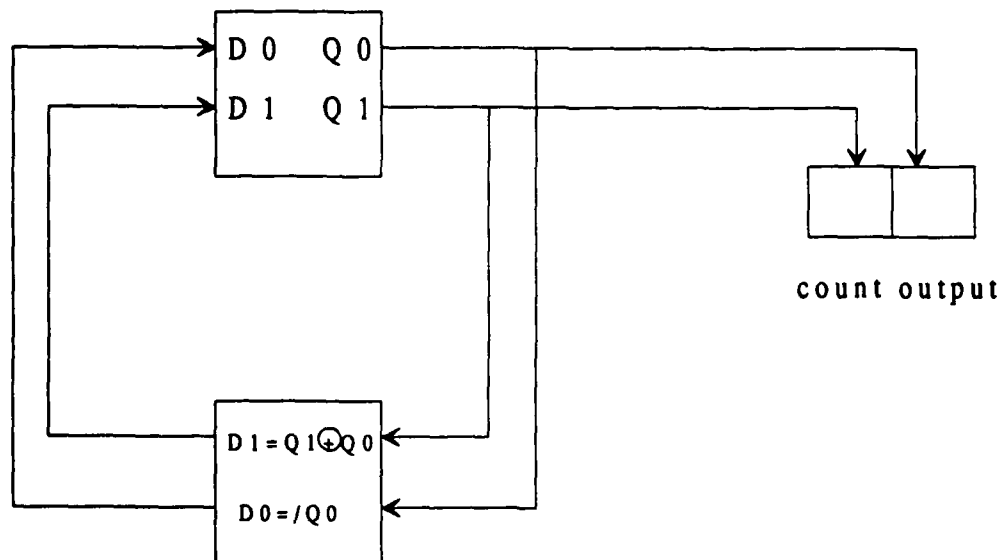
An arbitrary example (a counter) which fits the classic model, with well-defined discrete state variables

A discrete model -- does not require numerical integration

All "next states" are explicit functions of present states

Concept can be extended to simulations of systems of high complexity

Example Digital State Machine: Two-bit counter



The Two-bit Counter

Two state variables: Q1 and Q0

When viewed as a single two-bit number Q1Q0, counter will count 0, 1, 2, 3 (00, 01, 10, 11)

"Next-state" variables for Q1 and Q0 are called D1 and D0, respectively

Next-state equations are easily determined to be

$$D1 = Q1 \text{ exor } Q0$$

$$D0 = \text{not } Q0$$

Pseudo-code for Two-bit counter

Processor 0:

```
Q0 := 0 /* initialize */
while true do begin /* main loop */
  send (Q0)
  D0 := not Q0
  Q0 := D0 /* state assignment separated for clarity */
end
```

Processor 1:

```
Q1 := 0 /* initialize */
while true do begin /* main loop */
  receive (Q0)
  D1 := Q0 exor Q1
  Q1 := D1 /* state assignment separated for clarity */
end
```

A More General Discrete System

Another arbitrary example -- a second-order digital filter

Basically the same as digital logic example, except that variables and functions are not Boolean

Still a discrete model -- does not require numerical integration

All "next states" are still explicit functions of present states

Concept can be extended to simulations of digital control, digital signal processing, any discrete-time system (including nonlinear ones)

Digital Filter Example

$u(n)$ is input sequence, $y(n)$ is output sequence

$$\frac{Y(z)}{U(z)} = E(z) = \frac{z - 0.5}{z^2 - z + 0.5}$$

OR

$$y(n) = u(n-1) - 0.5 u(n-2) + y(n-1) - 0.5 y(n-2)$$

OR

$$y(n) = y(n-1) - 0.5 x(n-1) + u(n-1)$$

where $x(n) = y(n-1) + u(n-1)$

This final form gives the second-order system as two coupled first-order systems and is more useful for parallel execution

Pseudo-code for Digital filter example

Processor 0: (input function)

```

u := 1 /* initialize */
while true do begin /* main loop */
  send (u)
  /* just make the input an impulse function for now */
  u := 1
end

```

Processor 1: (calculates x)

```

x := 0 /* initialize */
while true do begin /* main loop */
  receive (u)
  receive (y)
  send (x)
  nextx := y + u
  x := nextx /* state assignment separated for clarity */
end

```

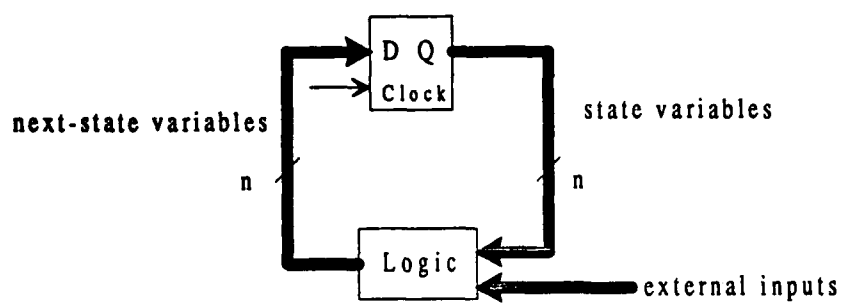
Processor 2: (calculates y)

```

y := 0 /* initialize */
while true do begin /* main loop */
  receive (u)
  send (y)
  receive (x)
  nexty := y - 0.5 * x + u
  y := nexty /* state assignment separated for clarity */
end

```

General Digital State Machine



**Pseudo-code for Simple
Harmonic Oscillator
example**

Processor 0: (input function -- a step)

```

u := 1 /* initialize */
t := 0
dt := 0.005
u := step(t)
while ( t < tfinal ) do begin /* main loop */
  send (u)
  u := step(t)
  t := t + dt
end

```

Processor 1: (calculates xd)

```

xd := 0 /* initialize */
t := 0
dt := 0.005
while ( t < tfinal ) do begin /* main loop */
  send (xd)
  receive (u)
  receive (x)
  xdd := u - ( b*xd + k*x)
  xd := integrate (xd, xdd, dt)    /* use integration routine from library */
  t := t + dt
end

```

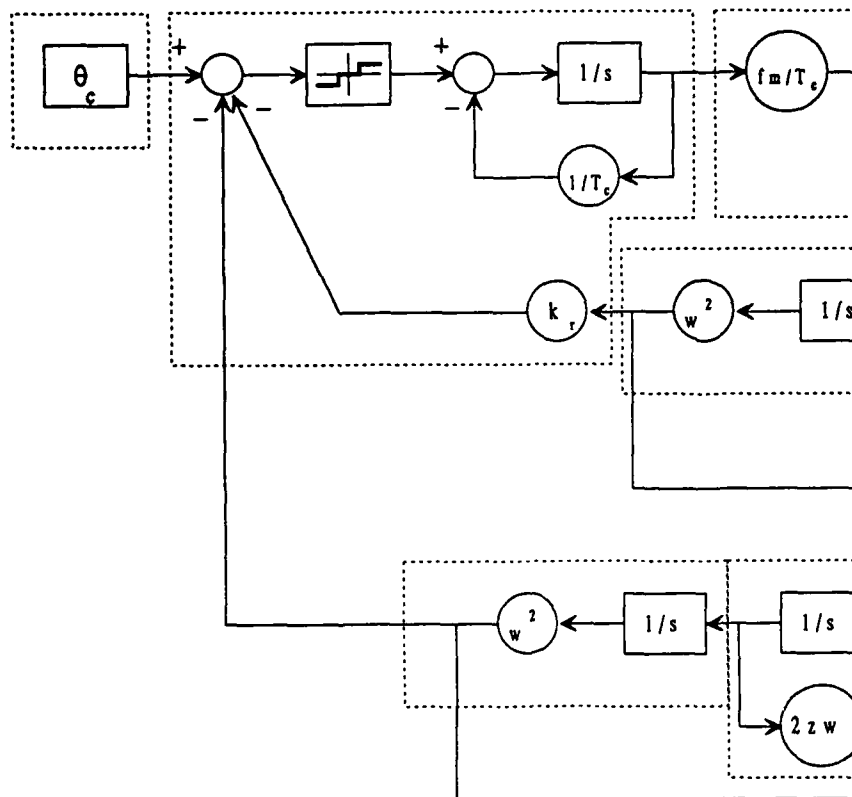
Processor 2: (calculates x)

```

x := 0 /* initialize */
t := 0
dt := 0.005
while ( t < tfinal ) do begin /* main loop */
  receive (xd)
  send (x)
  x := integrate (x, xd, dt)
  t := t + dt
end

```

Modified Satellite Attitude



program on their own workstation, then partition the Fortran code generated by the ACSL translator. This partitioning is semi-automated at present and can probably be made fully-automated.

The focus of the parallel programming process is the DERIVATIVE subsection of the DYNAMIC section in the ACSL program. The ACSL programmer begins by identifying sections of code which may be performed in parallel in the ACSL code by surrounding them with a PROCEDURAL statement and a matching END statement, even if some of these sections only include one line. Normally, these sections correspond to a functional unit, such as the IMU or a propulsion system. In the PROCEDURAL statement, all inputs and outputs must be listed. The ACSL compiler does not check these inputs and outputs for correctness in terms of the actual statements within the block. Sometimes ACSL programmers use this to their advantage to force the compiler to sort statements in a certain order, but this cannot be done if the program is eventually to be run in parallel. This is because we cannot allow the sort order to matter (outside of PROCEDURAL blocks, which remain unsorted internally).

Any statements which perform integration should not be placed inside PROCEDURAL blocks. These include the INTEG, INTVC, and LIMINT statements. The integration statements may be grouped together at the end of the program or each may be placed immediately after the PROCEDURAL block which calculates the derivative of the variable being integrated.

The result of this process should be a program in which all statements (except integrations) in the DERIVATIVE section belong to a PROCEDURAL block. One and only one PROCEDURAL block will have any given variable as an output -- the translator will enforce this. This allows the translator to sort the PROCEDURAL blocks in absolutely any order, while retaining the exact ordering of statements within each block. Since the simulation will run correctly with any ordering, it will also run correctly in parallel. The inputs and outputs given in the PROCEDURAL statements correspond to variables which are received or sent by that process over the crossbar interconnection network, so once again the importance of accuracy in these input/output lists becomes clear: a process will not have access to the required variables if they are not listed.

Each PROCEDURAL block is equivalent to an ADA task, and the input/output list specifies the required communications between tasks. Consequently, the PROCEDURAL ACSL implementation can be viewed as a step in the migration to ADA.

A more subtle aspect of the PROCEDURAL definitions is that, preferably, each PROCEDURAL block should output only derivative variables (i.e., variables which occur as the derivative in some integration statement). This allows each block to run in parallel during the two primary phases of each timestep: 1) derivative evaluation, and 2) integration. This is not a rigid requirement, and it can be worked around during the porting process.

6.2.2. Converting FORTRAN programs

Some programs, like EXOSIM, have already been written in FORTRAN, and some effort will be required to convert them to ACSL. This is not too difficult for several reasons. First, all

FORTTRAN subroutines are usable in an ACSL model, probably with no changes. Second, it is actually possible to eliminate some FORTRAN code, since integration is built into ACSL (and corresponding routines exist for the PFP). Finally, if the FORTRAN program is inherently modular, it should translate directly into the PROCEDURAL sections described above.

6.2.3. An Example Program

The example program to be presented here is modified in several stages to illustrate the major points. The result of each step is given as a listing of the ACSL model and is included in the Appendices. The ACSL program "missil.csl" is taken directly from the examples given in the ACSL manual. It implements a simple 6-DOF missile with only the basic functional elements. It has no target model, seeker, guidance law, or autopilot, but it is sufficient to illustrate the method.

A block diagram of the model is given as Figure 1. The dotted lines indicate the four partitions which are identified in the following section.

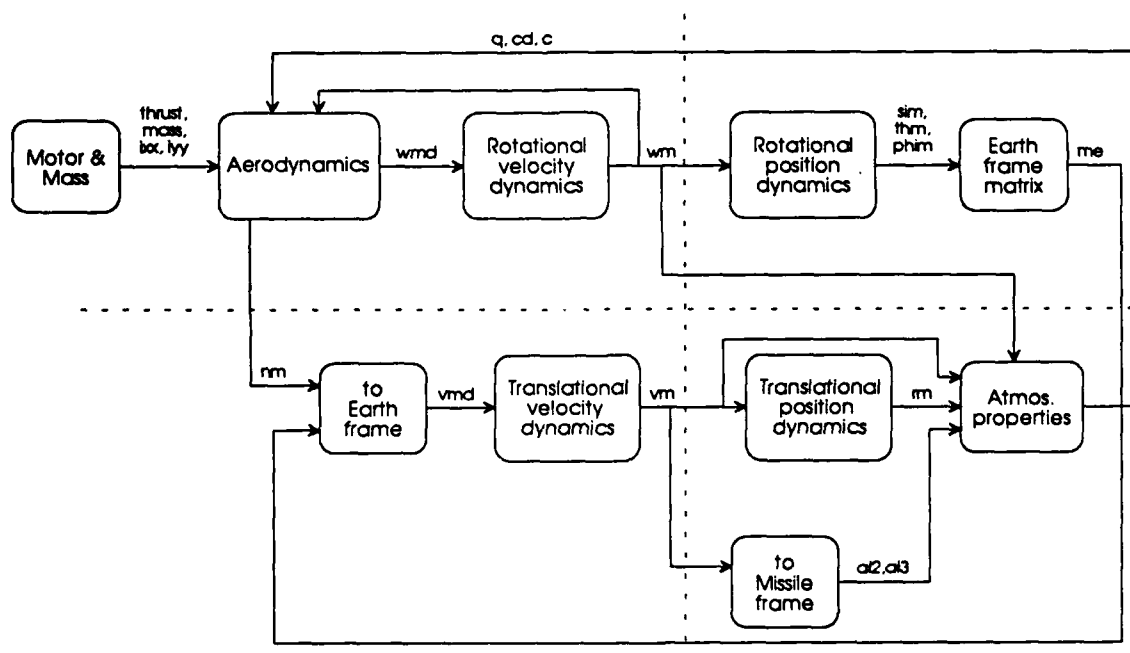


Figure 1: Block diagram of simple missile model
(reference: ACSL Reference Manual, Mitchell and Gauthier Associates)

6.2.3.1. Defining the main blocks

The statements of missil.csl were rearranged, and PROCEDURAL statements were added to form four main blocks:

- Block 0: motor, aerodynamics, and rotational velocity dynamics
- Block 1: rotational position dynamics

Block 2: translational velocity dynamics

Block 3: translational position dynamics

This particular partitioning is often effective for 6-DOF models. While the individual degrees of freedom may be further split out into smaller parallel blocks, it is often not advantageous, since there is much dependency on certain intermediate variables such as coordinate transformation matrices and aerodynamic coefficients.

The result of this partitioning is `missil2.csl`, also in the appendix. See the comments in the header and note that all changes to the original program are made in lower case. It is clear that very few changes were made, just some minor rearrangements (and deletions of the original PROCEDURALS, whose orderings were retained within the new PROCEDURALS).

There was no expectation that `missil2.csl` would run, however, since a circular definition exists. Specifically, by looking just at the PROCEDURAL statements (which is all that ACSL does to determine sort order for this model), it is obvious that NM depends on C, and C in turn depends on NM. This can be easily rectified by taking the statements which define NM(1), NM(2), and NM(3) out of block 0 and putting them in block 2, where they more logically belong. This is shown in the next stage, `missil3.csl`. This was not done earlier simply because it involved separating statements that were together in one of the example's original PROCEDURAL blocks. A quick inspection of the original program shows that there is no reason why these statements need to precede the statements which define WMD(1), WMD(2), and WMD(3).

Unlike the intermediate step (`missil2.csl`), `missil3.csl` will translate, compile, and run, giving the same results as the original model and taking the same time to execute. The difference is that the FORTRAN code generated by ACSL can be ported to four processors on the PFP.

6.2.3.2. Emphasizing state variables

While this four-processor implementation is usable, it would involve some additional manipulation, probably manually, in order to set up the appropriate communication channels. The problem is that `missil3.csl` does not adhere to the model of generating only derivative variables as outputs of each PROCEDURAL block. (A corollary to this is that all such blocks must depend only on state variables, or perhaps on other derivative variables).

The next stage, shown in `missil4.csl`, shows that it is possible to make this missile model adhere to the desired form. This usually involves replicating some code sections that generate convenient intermediate non-state variables that are used in more than one block (in this case, C, CD, and Q). This probably adds little or nothing to the execution time of the parallel implementation on the PFP in this case, since the replicated lines are added to block 0, which was relatively simple (block 2 determined the execution time). After the replication, blocks 0 and 2 are roughly equal in complexity and one or the other will determine the execution time. This model will also compile and give the same results, the only difference being that it ports seamlessly to the PFP.

As noted in the comments of `missil4.csl`, there is a simple way to make a slight improvement in execution time, left as an exercise for the reader.

6.2.3.3. Cheating time

One may think that the parallel implementation is not "correct" in the same sense that the serial implementation is correct, since it relies on "old" data that can be made available to all processors at the start of each integration step. This is simply not the case! The parallel implementation is a perfect implementation of proper numerical integration techniques applied to continuous systems for the purpose of digital simulation. As such, it is correct in the same sense as any corresponding serial implementation and will yield exactly the same results, assuming that computational precision is held constant.

The reason for this is that the parallel model relies only upon knowledge of the state variables at the beginning of each timestep. State variables are those variables which are integrated and thus contain the entire "memory" of the system. Intermediate algebraic variables are calculated on each processor as needed, and sometimes these calculations are replicated, as noted above, to eliminate the need for complex staging of communications.

It is possible, however, to force a little more parallelism into a model by allowing the use of "old" data. This will almost invariably affect the simulation results, so it must be done with discretion. The best candidates for this trick are sections of code which are fairly complex, generating intermediate variables which change relatively slowly over time. As an example, we will choose the sections of our missile model which calculate atmospheric damping and aerodynamic coefficients.

These sections are pulled out of the same lines of code which were replicated earlier on both block 0 and block 2. Since they contribute to the heavy processing on these blocks, they are reasonable candidates for pulling into separate blocks. It also seems reasonable that these coefficients would not change too much from one time step to the next.

The trick is shown in `missil5.csl`, in which two new blocks (4 and 5) are created. Each generates some new "derivative" variables, which are always set to zero. Then, when the "integration" is performed on the false state variables (C, CD, and Q), their values do not change from what they were set to inside the block. The result of this is that any block which needs C, CD, or Q gets their values one step late, but the calculation of these variables takes place in parallel.

This model also compiles and runs, but the results are different, as expected. If one plots the missile positions and velocities, there are no obvious differences, but listings of actual values for some variables show that the new model is not identical. When there is any doubt as to the acceptability of these deviations, this "trick" should not be implemented, and model state variables should correspond to true state variables in a physical sense.

6.2.3.4. Reaping the benefits

It is possible, though, to keep the original model and still reap many more benefits from parallel processing. This is accomplished by taking this simple model and gradually adding more components. Most such components include their own state variables and depend only on other state variables. A seeker model, for example, would generate some representation of what the seeker is seeing (the seeker state) based upon the relative positions of the target and the interceptor, which are also states. Since there is invariably some delay between the actual imaging process and the output of seeker data, there is no problem associated with having to use the current position states to determine the "next" seeker state. In fact, the delay is normally much longer than the integration step, so the programmer would most likely deliberately insert more delay in order to make a good seeker model.

Another example of a component which can be easily added is an IMU, which generates estimated missile states, generally based on current accelerometer states (which are not the actual accelerations).

Reasonable seeker and IMU models can be added to the example with absolutely no increase in execution time, since they can be performed in parallel on two or more additional processors.

A more difficult subsystem to add would be the accelerometers themselves, which must generate the required acceleration estimates for the IMU. The reason for this is that they are most easily modeled as a procedure which generates acceleration estimates based on actual accelerations, but actual accelerations are not states. (In this discussion, we are referring only to rotational accelerations and velocities, of course.) If we arbitrarily declare the actual accelerations to be states and then use them as inputs to an accelerometer block, then we are introducing a delay of one integration time step, which does not accurately model the real system. While this may be tolerable, it is more reasonable to simply add the accelerometer model to the same block (or blocks) which calculate the true accelerations. This block would then output estimated accelerations as new states, in addition to whatever states it already generated. (Previously, this block probably just integrated the actual accelerations to generate the actual velocities, which it outputted as states. Now it would output both velocities and estimated accelerations.)

In a complex missile model, there will be many other systems which can be added as new parallel blocks, like the seeker and IMU above. There will also be a few like the accelerometers, which must be incorporated into existing blocks in order to maintain an accurate model. If certain blocks grow to the point where they force execution time to grow to an unacceptable level, it may be possible to partition them by other techniques. The emphasis here is on a basic way of partitioning the major elements in such a way that a serial implementation can be transferred almost effortlessly to a parallel implementation which can be fine-tuned later, if necessary.

6.2.4. Summary

A proposal has been made that all simulations intended to be run on the PFP in the near future be written in ACSL, with some specific guidelines for structure. This has several advantages, including:

1. Specific identification of parallelism
2. Automatic translation to the PFP
3. Migration path to ADA
4. Built-in integration methods (in ACSL and on the PFP)
5. Support for specialized simulation functions (also in ACSL and on the PFP).

7. PFP File Formats

This section discusses the special file formats required by the PFP development tools. These include the "ENVIRONMENT" file, which specifies the system hardware configuration, the "PROCESS.TXT" file, which describes the processor configuration for a specific application, and the "NETWORK.TXT" file, which describes the network configuration for a specific application. All applications must have a "PROCESS.TXT" file, but only multiprocessor applications that communicate over the crossbar network (as is generally the case) require a "NETWORK.TXT" file.

7.1. Environment

The "ENVIRONMENT" file contains information necessary for mapping symbolic names used by the PFP development tools to actual hardware. It is a text file, and each line contains either information about a hardware element (crossbar, sequencer, or target processor) or a comment (always with a "#" as the first character on the line). Example 25 shows a full 32-processor configuration. Normally, the ENVIRONMENT file does not need to be altered by the programmer. It may be necessary to do so, however, if some processors are removed for service or if memory settings are changed.

The form of a non-comment line in the ENVIRONMENT file is:

```
<element name> = <base address>;<limit address>;<element type>;
```

where <element name> is the label used by other applications to refer to that element, <base address> is the starting memory address of the element in the host address space, <limit address> is the number of valid memory locations (in bytes), and <element type> is one of several valid element types. Currently the only element types supported are 28612, 38612, 0001, 0002, effe, and fffe. Two of these are processor types (28612 and 38612), two are for the "first" crossbar and sequencer (0001 and effe, respectively), and two are for the "second" crossbar and sequencer in a 64-processor system (0002 and fffe, respectively). All numeric fields are hexadecimal.

The <element name> field can be any 16-character string, as long as there are no repetitions. These names are used in the PROCESS.TXT and NETWORK.TXT files for each application, so they should usually not be changed from their default values (or else some applications will cease to run correctly).

The <base address> field is not actually a true physical address. Only the last six hex digits represent the memory address of each element. The first two hex digits are used to "turn on" the appropriate card cage, since there are at least four active card cages in a PFP, all mapped to the same address space but with no more than one enabled at any given time. This is done by issuing a particular I/O command to the address 8XX, where the X's are the first two digits. All of this is transparent to the programmer, so the eight-digit address can be viewed as a virtual address.

Note that the example is for the "second" half of a 64-processor system. The first half would contain processors p00 through p31.

Example 25: ENVIRONMENT.

```
# network 2 configuration
crossbar = 00040000;010000;0002;
sequencer = 00000000;010000;ffff;
# upper right bank configuration
p58 = 02100000;100000;28612;
p33 = 02200000;100000;28612;
p37 = 02300000;100000;28612;
p48 = 02400000;100000;28612;
p52 = 02500000;100000;28612;
p47 = 02600000;100000;28612;
p43 = 02700000;100000;28612;
p63 = 02800000;100000;28612;
p59 = 02900000;100000;28612;
p32 = 02a00000;100000;28612;
p36 = 02b00000;100000;28612;
# middle right bank configuration
p45 = 04100000;100000;28612;
p41 = 04200000;100000;28612;
p61 = 04300000;100000;28612;
p57 = 04400000;100000;28612;
p34 = 04500000;100000;28612;
p38 = 04600000;100000;28612;
p49 = 04700000;100000;28612;
p53 = 04800000;100000;28612;
p46 = 04900000;100000;28612;
p42 = 04a00000;100000;28612;
p62 = 04b00000;100000;28612;
# lower right bank configuration
p51 = 06100000;100000;28612;
p55 = 06200000;100000;28612;
p44 = 06300000;100000;28612;
p40 = 06400000;100000;28612;
p60 = 06500000;100000;28612;
p56 = 06600000;100000;28612;
p35 = 06700000;100000;28612;
p39 = 06800000;100000;28612;
p50 = 06900000;100000;28612;
p54 = 06a00000;100000;28612;
# <element name> = <base address>;<limit address>;<element type>;
```

7.2. Process.txt

The "PROCESS.TXT" file contains information about which processors are being used in this simulation, what program to load in each processor, where to get output data, and where to put output data. Comments are again indicated by a "#" in the first character position.

The format of each line is

```
<element name> <loadfile name> <inputfile name> <outputfile name>
```

where <element name> is a valid name from the ENVIRONMENT file, <loadfile name> is an object file name to be loaded to the element before starting each element, <inputfile name> is either a valid file name for input data or "<null>" and <outputfile name> is either a valid file name for output or "<null>". If "<null>" is used in the output file position, all output data is sent to the user terminal display. If "<null>" is used in the input file position, all input data is taken from the user keyboard. This routing of input data and output data is performed by the

IOSERVE program, so the final two fields may not have meaning for an application that does not use IOSERVE. (They should be present, however, at least as <null> entries.) The PROCESS.TXT file is also used by the RESET, DOWNLOAD, and START programs.

Example 26 shows a typical PROCESS.TXT file. In this application, the file "target/ntest.bl" is downloaded to each target processor, and each processor pXX receives input data from a file pXX.txt. The crossbar and sequencer load files are named "crossbar.bl" and "sequencer.bl."

Example 26:PROCESS.TXT.

```
# network 2
crossbar crossbar.bl <null> <null>
sequencer sequencer.bl <null> <null>
# upper right bank
p58 target/ntest.bl p58.txt <null>
p33 target/ntest.bl p33.txt <null>
p37 target/ntest.bl p37.txt <null>
p48 target/ntest.bl p48.txt <null>
p52 target/ntest.bl p52.txt <null>
p47 target/ntest.bl p47.txt <null>
p43 target/ntest.bl p43.txt <null>
p63 target/ntest.bl p63.txt <null>
p59 target/ntest.bl p59.txt <null>
p32 target/ntest.bl p32.txt <null>
p36 target/ntest.bl p36.txt <null>
# middle right bank
p45 target/ntest.bl p45.txt <null>
p41 target/ntest.bl p41.txt <null>
p61 target/ntest.bl p61.txt <null>
p57 target/ntest.bl p57.txt <null>
p34 target/ntest.bl p34.txt <null>
p38 target/ntest.bl p38.txt <null>
p49 target/ntest.bl p49.txt <null>
p53 target/ntest.bl p53.txt <null>
p46 target/ntest.bl p46.txt <null>
p42 target/ntest.bl p42.txt <null>
p62 target/ntest.bl p62.txt <null>
# lower right bank
p51 target/ntest.bl p51.txt <null>
p55 target/ntest.bl p55.txt <null>
p44 target/ntest.bl p44.txt <null>
p40 target/ntest.bl p40.txt <null>
p60 target/ntest.bl p60.txt <null>
p56 target/ntest.bl p56.txt <null>
p35 target/ntest.bl p35.txt <null>
p39 target/ntest.bl p39.txt <null>
p50 target/ntest.bl p50.txt <null>
p54 target/ntest.bl p54.txt <null>
```

7.3. Network.txt

The "NETWORK.TXT" file contains information about processor to processor communication. The format of this file retains compatibility with earlier versions of the PFP software and consists of a list of communication commands. Only two command types exist. The basic command is the CYCLE statement. The form of this statement is

```
CYCLE[<n>]
<processor_list> := <processor.repeat_count>
```

where <n> is a sequence number, <processor_list> is a set of processor element names separated by commas (the receiving processors), and <processor.repeat_count> is a processor element

name (the sending processor, optionally followed by a period and a repeat count. The repeat count simply causes multiple sixteen-bit transfers to take place. A double-precision real number, for example, will require a repeat count of 4.

The only other valid statement is the LOOP statement:

LOOP;

This indicates that all remaining cycles are performed in a repetitive loop as long as the processors continue to run. There is no means of breaking out of this loop, but any processor can terminate the application.

Example 27 shows the NETWORK.TXT file corresponding to the previous PROCESS.TXT example.

Example 27: NETWORK.TXT.

```

LOOP;
CYCLE [ 1 ]
p33, p37, p48, p52, p47, p43, p63, p59, p32, p36, p45, p41, p61, p57, p34, p38,
p49, p53, p46, p42, p62, p51, p55, p44, p40, p60, p56, p35, p39, p50, p54 := p58;
CYCLE [ 2 ]
p58, p37, p48, p52, p47, p43, p63, p59, p32, p36, p45, p41, p61, p57, p34, p38,
p49, p53, p46, p42, p62, p51, p55, p44, p40, p60, p56, p35, p39, p50, p54 := p33;
CYCLE [ 3 ]
p58, p33, p48, p52, p47, p43, p63, p59, p32, p36, p45, p41, p61, p57, p34, p38,
p49, p53, p46, p42, p62, p51, p55, p44, p40, p60, p56, p35, p39, p50, p54 := p37;
CYCLE [ 4 ]
p58, p33, p37, p52, p47, p43, p63, p59, p32, p36, p45, p41, p61, p57, p34, p38,
p49, p53, p46, p42, p62, p51, p55, p44, p40, p60, p56, p35, p39, p50, p54 := p48;
CYCLE [ 5 ]
p58, p33, p37, p48, p47, p43, p63, p59, p32, p36, p45, p41, p61, p57, p34, p38,
p49, p53, p46, p42, p62, p51, p55, p44, p40, p60, p56, p35, p39, p50, p54 := p52;
CYCLE [ 6 ]
p58, p33, p37, p48, p52, p43, p63, p59, p32, p36, p45, p41, p61, p57, p34, p38,
p49, p53, p46, p42, p62, p51, p55, p44, p40, p60, p56, p35, p39, p50, p54 := p47;
CYCLE [ 7 ]
p58, p33, p37, p48, p52, p47, p63, p59, p32, p36, p45, p41, p61, p57, p34, p38,
p49, p53, p46, p42, p62, p51, p55, p44, p40, p60, p56, p35, p39, p50, p54 := p43;
CYCLE [ 8 ]
p58, p33, p37, p48, p52, p47, p43, p59, p32, p36, p45, p41, p61, p57, p34, p38,
p49, p53, p46, p42, p62, p51, p55, p44, p40, p60, p56, p35, p39, p50, p54 := p63;
CYCLE [ 9 ]
p58, p33, p37, p48, p52, p47, p43, p63, p32, p36, p45, p41, p61, p57, p34, p38,
p49, p53, p46, p42, p62, p51, p55, p44, p40, p60, p56, p35, p39, p50, p54 := p59;
CYCLE [ 10 ]
p58, p33, p37, p48, p52, p47, p43, p63, p59, p36, p45, p41, p61, p57, p34, p38,
p49, p53, p46, p42, p62, p51, p55, p44, p40, p60, p56, p35, p39, p50, p54 := p32;
CYCLE [ 11 ]
p58, p33, p37, p48, p52, p47, p43, p63, p59, p32, p45, p41, p61, p57, p34, p38,
p49, p53, p46, p42, p62, p51, p55, p44, p40, p60, p56, p35, p39, p50, p54 := p36;
CYCLE [ 12 ]
p58, p33, p37, p48, p52, p47, p43, p63, p59, p32, p36, p41, p61, p57, p34, p38,
p49, p53, p46, p42, p62, p51, p55, p44, p40, p60, p56, p35, p39, p50, p54 := p45;
CYCLE [ 13 ]
p58, p33, p37, p48, p52, p47, p43, p63, p59, p32, p36, p45, p61, p57, p34, p38,
p49, p53, p46, p42, p62, p51, p55, p44, p40, p60, p56, p35, p39, p50, p54 := p41;
CYCLE [ 14 ]
p58, p33, p37, p48, p52, p47, p43, p63, p59, p32, p36, p45, p41, p57, p34, p38,
p49, p53, p46, p42, p62, p51, p55, p44, p40, p60, p56, p35, p39, p50, p54 := p61;
CYCLE [ 15 ]
p58, p33, p37, p48, p52, p47, p43, p63, p59, p32, p36, p45, p41, p61, p34, p38,
p49, p53, p46, p42, p62, p51, p55, p44, p40, p60, p56, p35, p39, p50, p54 := p57;
CYCLE [ 16 ]
p58, p33, p37, p48, p52, p47, p43, p63, p59, p32, p36, p45, p41, p61, p57, p38,
p49, p53, p46, p42, p62, p51, p55, p44, p40, p60, p56, p35, p39, p50, p54 := p34;

```


[illegible]

8. Appendices

- 8.1. C Program Library
- 8.2. FORTRAN Program Library
- 8.3. Pascal Program Library
- 8.4. PL/M Program Library
- 8.5. Sample ACSL files